Reading: Chapter 07 of [Chiusano and Bjarnason 2014]

Purely Functional Parallelism

Lecture 060 of Advanced Programming

March 7, 2018

Andrzej Wasowski & Zhoulai Fu

IT University of Copenhagen

How today's class differs from your PCPP (Practical Concurrent and Parallel Programming) class

- PCPP is about writing efficient concurrent and parallel software, using existing API in Java
- Today's class is about *designing* such API, under the functional paradigm
- A minor note about concurrency vs parallelism:
  - Concurrency describes a *problem* — that things needs to happen together
  - Parallelism describes a *solution* — *that is* based on multiple threads/CPUs
  - Other solutions exist; we focus on parallelism today

Prerequisite 1: ExecutorService and Future API

```
class ExecutorService {
  def submit[A](a: Callable[A]): Future[A]
}
trait Future[A] {
  def get: A
}
```

```
public interface ExecutorService
extends Executor
```

An `Executor` that provides methods to manage termination and methods that can produce a `Future` for tracking progress of one or more asynchronous tasks.

DEMO (if time allows)

```
val es=Executors.newWorkStealingPool()
val fut=es.submit(new Callable[Int] {
  override def call(): Int = (1 to 10).sum
})
val res=fut.get
```

Prerequisite 2: Strict and Lazy Functions

- Strict Evaluation (by-value): function argument evaluated before entering the function
- Lazy evaluation, (by-name, by-need)

DEMO (if time allows)

```
def f_lazy(i: => Int) = {println("Entering call");i}
def f_strict(i: Int) = {println("Entering call");i}

f_lazy{println("Calling"); 42} // Entering call, calling

f_strict{println("Calling"); 42} // Calling, Entering call
```

Designing the API

- No right answers in design
- You will see a collection of design choices
- You are to understand their trade-offs, and think critically.

Why not use Java Thread

```
trait Runnable { def run: Unit }

class Thread(r: Runnable) {
  def start: Unit
  def join: Unit
}
```

Begins running r in a separate thread.

Blocks the calling thread until r finishes running.

- **Side-effects are evil when it comes to program reasoning:** if we want to get any information out of a `Runnable`, it has to have some side effect, like mutating some state that we can inspect. This is bad for compositionality—we can't manipulate `Runnable` objects generically since we always need to know something about their internal behavior.
- **Thread is low-level and low-efficient:** `Thread` maps directly onto operating system threads, which are a scarce resource. It would be preferable to create as many "logical threads" as is natural for our problem, and later deal with mapping these onto actual OS threads.

Design Goals

- **Pure**: function return the same value for the same input, without observable side effects
- **High-level**: having the capability to write something like foldleft, as in sequential programs.

```
def sum(ints: Seq[Int]): Int =
  ints.foldLeft(0)((a,b) => a + b)
```

Design Methodologies

- Start from simple examples
- Try - Challenge - Refine

Example: Summing a list with divide-and-conquer

```scala
def sum(ints: IndexedSeq[Int]): Int =
  if (ints.size <= 1)
    ints.headOption getOrElse 0
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    sum(l) + sum(r)
  }
```

`IndexedSeq` is a superclass of random-access sequences like `Vector` in the standard library. Unlike lists, these sequences provide an efficient `splitAt` method for dividing them into two parts at a particular index.

Divides the sequence in half using the `splitAt` function.

`headOption` is a method defined on all collections in Scala. We saw this function in chapter 4.

Recursively sums both halves and adds the results together.

- Listing 7.1, [Chiusano et al]

The Making of a Parallel Sum (1 - try)

```
def sum(ints: IndexedSeq[Int]): Int =
  if (ints.size <= 1)
    ints headOption getOrElse 0
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    val sumL: Par[Int] = Par.unit(sum(l))
    val sumR: Par[Int] = Par.unit(sum(r))
    Par.get(sumL) + Par.get(sumR)
  }
```

Computes the left half in parallel.

Computes the right half in parallel.

Extracts both results and sums them.

- Need a data type to contain parallel computation results: **Par[A]**
- Need a function to evaluate a computation in a separate thread
  - **Par.unit** (a: =>A): Par[A]
- Need another function to extract a result from a Par[A]:
  - **Par.get [A] (a: Par[A]):A**

The Making of a Parallel Sum (1 - problem)

```
def sum(ints: IndexedSeq[Int]): Int =
  if (ints.size <= 1)
    ints headOption getOrElse 0
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    val sumL: Par[Int] = Par.unit(sum(l))
    val sumR: Par[Int] = Par.unit(sum(r))
    Par.get(sumL) + Par.get(sumR)
  }
```

Computes the left half in parallel.

Computes the right half in parallel.

Extracts both results and sums them.

- For the sake of parallelization, Par.unit has to delay the computation until Par.get
- Problem: The whole computation is still sequential because "+" is strict

The Making of a Parallel Sum: (2 - try)

```
def sum(ints: IndexedSeq[Int]): Par[Int] =
  if (ints.size <= 1)
    Par.unit(ints.headOption getOrElse 0)
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    Par.map2(sum(l), sum(r))(_ + _)
  }
```

- Par.map2 is a new higher-order function for combining the result of two parallel computations.
- Q: What is its signature?
- A: Par.map2[A,B,C] (a: Par[A], b: Par[B]) (f: (A,B) => C): Par[C]
- Q: Should Par.map2 be lazy or strict?
- A: :If it is strict, we'll strictly construct the entire left half of the tree of summations first before moving on to (strictly) constructing the right half ==> Let Par.map2 be lazy

The Making of a Parallel Sum: (2 - problem)

```
def sum(ints: IndexedSeq[Int]): Par[Int] =
  if (ints.size <= 1)
    Par.unit(ints.headOption getOrElse 0)
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    Par.map2(sum(l), sum(r))(_ + _)
  }
```

- Q: Do we always want to evaluate the two arguments to Par.map2 in parallel?
- A: : Probably not. Consider Par.map2(Par.unit(1), Par.unit(2))(_+_). The overhead for thread creation/management is swamping any tiny gains from parallelization.
- Problem: This API is ver inexplicit about when computations gets forked off the main thread — the programmer cannot specify where this forking should occur.

The Making of a Parallel Sum: (3 - try)

```
def sum(ints: IndexedSeq[Int]): Par[Int] =
  if (ints.length <= 1)
    Par.unit(ints.headOption getOrElse 0)
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    Par.map2(Par.fork(sum(l)), Par.fork(sum(r)))(_ + _)
  }
```

- Par.fork[A](a: => Par[A]): Par[Int] runs *a* in a separate logical thread
- With *Par.fork*, you can make *Par.map2* strict, leaving it up to the programmer to wrap arguments if they want

The Making of a Parallel Sum: (3 - problem)

```scala
def sum(ints: IndexedSeq[Int]): Par[Int] =
  if (ints.length <= 1)
    Par.unit(ints.headOption getOrElse 0)
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    Par.map2(Par.fork(sum(l)), Par.fork(sum(r)))(_ + _)
  }
```

- Problem: If *Par.fork* runs *sum(l)* or *sum(r)* immediately in a separate thread, the thread pool (or whatever resource we use to implement the parallelism) must be globally accessible and properly initialized wherever we want to call — meaning that we lose the ability to control the parallelism strategy used for different parts of our program.
- Although there's nothing inherently wrong with having a global resource for executing parallel tasks, it seems more appropriate to give *get* the responsibility of creating threads and submitting execution tasks.

The Making of a Parallel Sum: (final)

- We let *fork* hold on to its unevaluated argument until later. It takes an unevaluated *Par[A]* and marks it for concurrent evaluation later
- In this model, *Par[A]* holds a *description* of a parallel computation that gets *interpreted* at a later time by something like the *get* function
- How to implement it?

Implementation: reused API from Java

```
class ExecutorService {
  def submit[A](a: Callable[A]): Future[A]
}
trait Callable[A] { def call: A }
trait Future[A] {
  def get: A
  def get(timeout: Long, unit: TimeUnit): A
  def cancel(evenIfRunning: Boolean): Boolean
  def isDone: Boolean
  def isCancelled: Boolean
}
```

- *ExecutorService* lets us submit a *Callable* value and get back a corresponding *Future,* which is a handle to a computation that's potentially running in a separate thread.
- When *Future* obtain a value from *get,* it blocks the current thread until the value is available.
- *Future* has some extra features for cancellation (throwing an exception after blocking for a certain amount of time, and so on).

Implementation: Other API

- Type alias: *type Par[A] = ExecutorService => Future[A]*
- Object Par that holds three primitive operations: *unit, map2, and fork*

Quiz

- Define map[A,B](pa: Par[A])(f: A => B): Par[B] in terms of:
    - map2[A,B,C] (a: Par[A], b: Par[B]) (f: (A,B) => C): Par[C]

Quiz

- Define map[A,B](pa: Par[A])(f: A => B): Par[B] in terms of:
  - map2[A,B,C] (a: Par[A], b: Par[B]) (f: (A,B) => C): Par[C]

- Solution:

```
  def map[A,B](pa: Par[A])(f: A => B): Par[B]
= map2(pa, unit(()))((a,_) => f(a))
```

- The fact that we can implement map in terms of map2 but not the other way around, just shows that map2 is strictly more powerful than map.
- This sort of thing happens a lot when we're designing libraries—often, a function that seems to be primitive will turn out to be expressible using some more powerful primitive.

Answer

- Define
  - map[A,B](pa: Par[A])(f: A => B): Par[B]
- in terms of:
  - map2[A,B,C] (a: Par[A], b: Par[B]) (f: (A,B) => C): Par[C]

```
def map[A,B](pa: Par[A])(f: A => B): Par[B] =
  map2(pa, unit(()))((a,_) => f(a))
```

Laws and Properties

- Consider a unit test (incidentally of the unit function):
  - map(unit(1)) (_ + 1) == unit(2)
- What does equality mean on the Par[Int] values?
- For instance with the following definition of equality:
  - def equal[A] (e: ExecutorService) (p: Par[A], p2: Par[A]) :Boolean = p(e).get == p2(e).get
- But how would we test map? A more general test would be nice:
  - map (pa) (f).get == f(pa.get) for all pa,f
- This is no longer a unit test, but a property.
- Next week you will look into how to turn such properties into tests systematically

Takeaway

- Not only how to write a library for purely functional parallelism, but *how to approach the problem of designing a purely functional library*.